UNIVERSIDAD AUTÓNOMA DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Clasificación por Análogos

Alberto Velázquez Sánchez
Tutor: Ángela Fernández Pascual
Ponente: José Ramón Dorronsoro Ibero

Junio 2018

Clasificación por análogos

AUTOR: Alberto Velázquez Sánchez
TUTOR: Ángela Fernández Pascual

Dpto. Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid

Junio de 2018

Resumen (castellano)

Actualmente en distintos ámbitos de la sociedad moderna se menciona el término *machine learning* o aprendizaje automático. El aprendizaje automático es una disciplina que puede generar un valor de negocio tangible para una amplia gama de industrias. En concreto, uno de los posibles problemas a abordar son los de clasificación, en los cuáles algunos de sus algoritmos necesitan el uso de medidas de disimilitud para que el algoritmo pueda ejecutar su clasificación, siendo la elección de una disimilitud u otra esencial para que el algoritmo lleve a cabo la clasificación de manera correcta.

Por ello nace el planteamiento de realizar un estudio y análisis de estos algoritmos de aprendizaje automático poniendo especial énfasis en el algoritmo de *k vecinos más próximos* que se trata de un método de clasificación supervisada. Debido a que en este método se pueden utilizar distintas distancias para su generación de modelos, en este estudio nos centraremos especialmente en la comparación del algoritmo utilizando la distancia euclídea y utilizando la distancia de Mahalanobis.

Este análisis se hace utilizando herramientas que nos proporciona la librería *open source* de aprendizaje automático llamada *scikit-learn* que está presente en el lenguaje de programación *Python*. Sin embargo, este análisis además ha derivado en la creación de un nuevo modelo de clasificación definido como clasificador por análogos, en el cual hemos tenido que implementar una nueva distancia basada en Mahalanobis, pero introduciendo un concepto de localidad. La codificación de esta distancia local se ha hecho mediante el lenguaje de programación *Python* y se integra a la perfección con la librería *scikit-learn*.

Finalmente, en el presente proyecto procedemos a realizar una serie de experimentos de tres modelos de *k vecinos más próximos*, cada uno con una de las tres distancias mencionadas, utilizando para ello distintos conjuntos de datos. Tras analizar los resultados, concluimos que el nuevo modelo de clasificación por análogos obtiene mejores resultados con respecto a los otros modelos analizados.

Palabras clave (castellano)

Machine learning, Disimilitud, K vecinos más próximos, Distancia euclídea, Distancia de Mahalanobis, Open source, Scikit-learn, Python.

Abstract (English)

Currently, in every field in the society it is mentioned the term machine learning. It is a discipline that can generate an accurate business value for a broad range of industries. In particular, one of those posible problems could be classification assignment That needs an appropiated dissimilarity measure to run the perfect classification depending on what we are looking for.

Therefore, we decide to study and analyse those algorithms of machine learning in relation with the *k nearest neighbors* algorithm, a supervised classification method. This method can be applied wih different distances, that will give rise to different models. In this study we are going to focus on the comparision between euclidean distance and Mahalanobis distance.

This analysis is made in *Phyton*, using the *open source* library called *scikit-learn*, which is specially design for machine learning purposes. However this analysis has derivated in the creation of a new classification model defined as classification model by analogs in which we had implemented a new distance based on Mahalanobis since a local perspective. The code generated in *Phyton* is perfectly integrated with the *scikit-learn* library.

Finally, we made some experiments for testing these three models over many datasets. After analylzing the results we conclude that the new classification model presented outperforms the other models.

Keywords (inglés)

Machine learning, Dissimilarity, K nearest neighbors, Euclidean distance, Mahalanobis distance, Open source, Scikit-learn, Python.

Agradecimientos

Primero quiero agradecer a mi tutora Ángela Fernández y al profesor Carlos M. Alaíz por todo lo que me han enseñado en este trabajo y por toda la ayuda recibida en él y además a todos los profesores que me han guiado durante la carrera y me han hecho aprender de manera satisfactoria.

A Pablo, Jorge, Luis y José que desde que tuvimos la fortuna de conocernos no nos hemos separado, hemos ido superando todos los retos que nos han ido apareciendo y hemos disfrutado de inolvidables momentos y los que quedan por disfrutar.

Dar las gracias a mis padres, a mis hermanos y a toda mi familia en general por todo el apoyo brindado, sin el que no hubiera sido posible llegar hasta aquí.

INDICE DE CONTENIDOS

INDICE DE FIGURAS

INDICE DE TABLAS

1 Introducción

1.1 Motivación

En 1950 Alan Turing, considerado como el padre de la Inteligencia Artificial crea el *Test de Turing*, el cual es una prueba de habilidad de una máquina para exhibir un comportamiento inteligente similar al de un ser humano o indistinguible de este. En 2016 la inteligencia Artificial de Google derrota a un jugador profesional en una partida del juego *Go*, que se considera el juego de mesa más complejo de todos.[1]

Dados estos hechos se intuye que la importancia de la inteligencia artificial en nuestra sociedad ha crecido a un ritmo vertiginoso en los últimos años, especialmente en el aprendizaje automático donde se desarrollan algoritmos que descubren conocimiento a partir de datos y experiencias específicas, basados en sólidos principios estadísticos y computacionales. De ahí la existencia de un gran interés por intentar entender cómo actúan estos algoritmos de aprendizaje automático y por cómo utilizarlos para explotar toda la información de la que disponemos hoy en día y sacar el mayor beneficio de ellos.

De hecho, actualmente nos encontramos en la denominada era del Big Data por el gran "boom de datos" generado por empresas y distintos sectores de la sociedad. Probablemente este boom es debido a la digitalización de gran parte de los datos y el abaratamiento de su almacenamiento y procesamiento. Gracias a esto, en los últimos años el interés por el concepto de *machine learning* o aprendizaje automático no para de crecer, ya que estos algoritmos son muy valiosos al permitir a organizaciones y empresas adelantarse a los hechos que puedan ocurrir, es decir, les ayuda a predecir. Esto puede apreciarse en la Figura 1, donde observamos el crecimiento de interés en búsquedas relacionadas con el término machine learning en los últimos cinco años mediante el motor de búsqueda *Google*, y en la Figura 2, donde se muestra en qué regiones del mundo se realiza una mayor búsqueda de este término. En la Figura 3 observamos la aparición de dicho término en libros publicados desde 1950, que se encuentran indexados por el motor de búsqueda *Google*.

Figura 1: Interés de búsqueda del término "Machine learning" en los últimos cinco años.**(**Fuente:[2])

Figura 2: Interés del término “Machine learning” por región. (Fuente:[2])

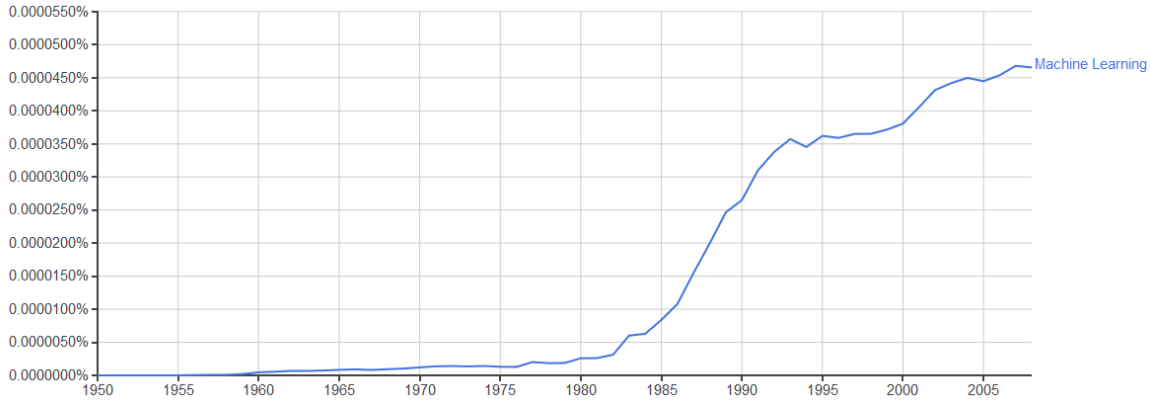

Figura 3: El término “machine learning” en libros. (Fuente: [3])

El *machine learning* o aprendizaje automático es una rama de la inteligencia artificial cuyo objetivo es el desarrollo de técnicas que permitan a las computadoras reproducir el mecanismo de aprendizaje que poseen los seres humanos a partir de información suministrada en forma de ejemplos, es por tanto un proceso de inducción al conocimiento [4]. Actualmente algunos de los usos de esta disciplina son el reconocimiento del lenguaje hablado y escrito, robótica, motores de búsqueda o detección de fraudes con el uso de tarjetas de crédito.

Existen números algoritmos de *machine learning* que se deben adaptar al problema en cuestión. Una primera distinción entre los algoritmos consiste en observar si se trata de un algoritmo de aprendizaje supervisado o no supervisado. Los algoritmos de aprendizaje supervisados son aquellos en los que, a partir de un conjunto de ejemplos clasificados (conjunto de entrenamiento), intentamos asignar una clasificación a un segundo conjunto de ejemplos (conjunto de test) de los cuales no conocemos la clase. En cambio, los algoritmos de aprendizaje no supervisado son aquellos en los que no disponemos de una batería de ejemplos previamente clasificados, sino que únicamente a partir de las propiedades de los ejemplos intentamos dar una agrupación (*clustering*) de los ejemplos según su similaridad.

Dentro del aprendizaje supervisado los dos tipos principales de métodos son los métodos de clasificación y de regresión, centrando el presente trabajo en los métodos de clasificación. Los métodos de clasificación ofrecen soluciones a los problemas de clasificación, los cuáles se centran en predecir a que “clase” o “etiqueta” pertenece un ejemplo [5].Hay una gran

cantidad de algoritmos de clasificación como Knn, regresión logística, arboles de decisión, Naive-Bayes, etc.

En estos métodos la distancia más común es la distancia euclídea, pero el uso de distancias menos usuales como la de Mahalanobis que cuantifican las diferencias entre objetos estadísticos, como variables aleatorias o distribuciones de probabilidad, tienen numerosas aplicaciones en campos como la biología, genética, psicología, etc. A la vista de esto cabe preguntarse si el uso de estas distancias podría mejorar el grado de precisión de los algoritmos para algunos problemas.

En este trabajo se va a proponer un nuevo método de clasificación el cual consiste en utilizar el algoritmo de clasificación knn, usando como distancia una variante de la distancia de Mahalanobis que combina las ventajas de esta distancia junto con las ventajas de la distancia euclídea. A este nuevo clasificador le llamaremos clasificador por análogos.

En cuanto al ámbito personal, me gustaría destacar el interés por desarrollar conocimientos en el campo del aprendizaje automático, área que durante la carrera ha sido mencionada en la asignatura de *Inteligencia Artificial* y *Fundamentos de Aprendizaje Automático*. De este modo, este proyecto se presenta como una oportunidad perfecta para desarrollar y ampliar estos conocimientos previos.

1.2 Objetivos

El principal objetivo de este TFG ha sido el estudio de los métodos de clasificación, concretamente del método knn, y del uso de la distancia euclídea y de Mahalanobis en este clasificador. Además, se propone un nuevo clasificador basado en una modificación de la distancia de Mahalanobis. Para ello se definen los siguientes objetivos:

- Analizar el uso de la distancia euclídea y la distancia de Mahalanobis en el algoritmo de clasificación knn.

- Diseñar una nueva distancia tomando como referencia la de Mahalanobis e implementarla para crear un nuevo clasificador.

- Realizar una serie de experimentos con el clasificador implementado comparándolo con dos clasificadores más, uno utilizando la distancia euclídea y otro la distancia de Mahalanobis tradicional.

- Analizar los resultados obtenidos para comprobar si la aproximación tomada mejora el rendimiento con respecto a las otras aproximaciones clásicas.

Cabe destacar como objetivo secundario el uso de la librería *scikit-learn* de *Python* como librería de *machine learning*, puesto que se implementa una nueva clase para desarrollar el nuevo clasificador y la nueva distancia propuestos mediante esta librería.

1.3 Organización de la memoria

La memoria consta de los siguientes capítulos:

- Introducción: En este apartado se explica tanto la motivación que propicia el estudio de este trabajo como los objetivos que pretendemos alcanzar en base a una serie de experimentos que se han llevado a cabo durante la realización del presente trabajo.

- Estado del arte: Este capítulo consta de cuatro puntos principales:
    - Se realiza una introducción al concepto de distancia.
    - Se define la distancia euclídea y se muestran sus ventajas y desventajas, así como sus posibles aplicaciones.
    - Se define que es la distancia de Mahalanobis, se destacan sus ventajas y desventajas, y se nombran algunas de sus aplicaciones actuales.
    - Se introduce el algoritmo de clasificación de k vecinos más próximos y se explica su funcionamiento.

- Diseño: A lo largo de este capítulo se explicará de manera detallada cómo se ha desarrollado la implementación del clasificador por análogos, qué diseño se ha seguido para su desarrollo y su implementación explicando brevemente las librerías utilizadas.

- Pruebas y resultados: En este apartado realizamos una serie de experimentos con distintos conjuntos de datos y con los resultados obtenidos sacamos una serie de conclusiones de nuestro clasificador por análogos.

- Conclusiones y trabajo futuro: Se realiza una breve recapitulación de los objetivos del trabajo, así como una última evaluación de los resultados obtenidos. Finalmente se detallan posibles formas de ampliar en el futuro el presente trabajo.

2 Estado del arte

2.1 Distancia

Una distancia δ sobre un conjunto X es una aplicación $X \times X \rightarrow \mathbb{R}$ tal que a cada par (a,b) le hace corresponder a un número real δ (a,b) = $\delta_{a,b}$. La distancia δ posee las siguientes propiedades [6]:

1. $\forall a, b \in X, \delta(a, b) \geq 0$

2. $\forall a \in X, \delta(a, a) = 0$

3. $\forall a, b \in X, \delta(a, b) = \delta(b, a)$

4. $\forall a, b, c \in X, \delta(a, b) \leq \delta(a, c) + \delta(c, b)$

5. $\forall a, b \in X, \delta(a, b) = 0 \Leftrightarrow a = b$

La distancia debe cumplir al menos las propiedades 1, 2 y 3. Si además cumple las propiedades 4 y 5 nos referimos a una *distancia métrica*. Al hablar de una distancia nos referiremos formalmente a una *distancia métrica*.

Existen una gran cantidad de distancias e indicadores de *disimilitud* y no se puede disponer de una regla general que nos permita definir una *disimilitud* conveniente para todo tipo de análisis. La adecuada elección de una u otra dependerá de múltiples factores como son las propiedades de que goce la distancia a utilizar, de la naturaleza de las variables utilizadas y de los individuos estudiados, o de la finalidad del análisis. La distancia más usual es la distancia euclídea, pero caben destacar otras distancias también muy utilizadas como la distancia de Mahalanobis, la distancia de Chebyshev o la distancia de Minkowski y su caso particular más conocido: la distancia de Manhattan. En este trabajo nos centraremos en la distancia de Mahalanobis y euclídea.

2.2 Distancia Euclídea

La distancia euclídea es la distancia ordinaria entre dos puntos en un espacio euclídeo. La base de esta distancia se encuentra en la aplicación del teorema de Pitágoras sobre triángulos rectos, donde la distancia euclidiana se define de la hipotenusa del triángulo recto conformado por cada punto y los vectores proyectados sobre los ejes directores al nivel de la hipotenusa.

2.2.1 Definición

La distancia euclídea d_E entre dos ejemplos del espacio euclideo dimensional se define como [7]:

$$d_E(X, Y) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \ldots + (x_n - y_n)^2} = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2} \qquad (2.1)$$

2.2.2 Ventajas e inconvenientes

La métrica euclídea pese a ser una de las distancias más simples y con mayor sencillez de cálculo que nos permite observar fácilmente propiedades de las distancias, presenta una serie de inconvenientes que hay que comentar.

Uno de ellos es que la métrica euclídea es una distancia sensible a las unidades de medida de variables. Las diferencias entre los valores de variables medidas con valores altos contribuirán en mayor medida que las diferencias entre valores de las variables con valores bajos. Esto da lugar a que sea una distancia muy sensible a los cambios de escala ya que se provocan cambios en las distancias entre puntos. Para evitar este problema se utiliza normalmente la distancia euclidiana normalizada o se recurre a la tipificación previa de las variables.

Otro problema que se nos presenta no proviene de esta distancia si no de la naturaleza de las variables. Si las variables que vamos a utilizar están correladas, estas variables nos darán en gran medida una información redundante debido a que parte de las diferencias entre los valores individuales de algunas variables podrían explicarse por las diferencias en otras variables. Como consecuencia de ello, la distancia euclídea inflará la *disimilitud* o divergencia entre los individuos. Una solución es analizar las componentes principales que están decorreladas en vez de las variables originales o ponderar la contribución de cada par de variables con pesos inversamente proporcionales a las correlaciones, lo que nos llevará a hablar de la distancia de Mahalanobis. Como resumen, la distancia euclídea es recomendable utilizarla cuando las variables sean homogéneas o cuando se desconozca la matriz de varianzas.

Sin embargo, a pesar de las desventajas que hemos mencionado, nos encontramos que se trata de la distancia más común y conocida. Este fenómeno se debe a que la distancia euclídea es la distancia estándar que se utiliza en algoritmos de clasificación, como el algoritmo knn que analizamos en este trabajo o de algoritmos de métodos de kernel como las máquinas de vectores de soporte (SVM). También es común su utilización en métodos de análisis de redes sociales.

2.2.3 Aplicaciones

La distancia euclídea se trata de la distancia estándar en muchos algoritmos de clasificación como hemos comentado en la sección anterior, pero además a pesar de su aparente simplicidad se utiliza en distintas aplicaciones como son las siguientes:

- Localización del sensor de una red [8].
- Estática de estructuras rígidas [8].
- Visualización de datos [8].
- Robótica[8].
- Conformación de proteínas a partir de datos RMN [9].
- Detección de comunidades en redes sociales [10].

2.3 Distancia Mahalanobis

La distancia de Mahalanobis es una medida de distancia introducida por Prasanta Chandra Mahalanobis en 1936. Su utilidad radica en qué se trata de una medida de distancia óptima para obtener la similitud entre dos puntos de una distribución ya que, a diferencia de la distancia euclídea, esta medida tiene en cuenta las características de las variables aleatorias que definen los atributos de cada punto mediante la matriz de covarianza. Esta distancia cumple las propiedades necesarias para ser una *distancia métrica.*

2.3.1 Definición

Formalmente la distancia de Mahalanobis d_M entre dos ejemplos (x e y) se define [11]:

$$d_M(x, y) = \sqrt{(x-y)^T \Sigma^{-1} (x-y)} \qquad (2.2)$$

Siendo Σ^{-1} la *matriz de covarianzas* del conjunto de datos. Si se sustituye esta matriz de covarianzas por la matriz identidad, entonces la distancia de Mahalanobis se transforma en la distancia euclídea.

2.3.2 Ventajas e inconvenientes

La distancia de Mahalanobis se basa en la matriz de covarianza de todas las variables y por lo tanto presenta ventajosas propiedades para solventar grandes inconvenientes mencionados anteriormente en la distancia euclídea.

En primer lugar, permite tomar en consideración la correlación del conjunto de datos en escala invariante, es decir, que no depende de una escala única de medida y además al considerar las correlaciones entre variables se corrige el efecto de la redundancia.

Esta distancia proporciona un poderoso método para saber si un determinado conjunto de condiciones similares es en realidad un conjunto de condiciones ideales, además de ser muy útil para identificar qué partes de un escenario son las más parecidas a las de un escenario ideal.

Sin embargo, esta distancia también presenta inconvenientes, siendo la más importante la relacionada con el cálculo de la matriz de covarianza, puesto que puede ser computacionalmente restrictiva dependiendo del problema o incluso no puede llegar a calcularse si las variables donde se quiere calcular están altamente correladas.

En la siguiente figura observamos la relación entre la distancia euclídea y la distancia de Mahalanobis. Las líneas representan puntos equidistantes respecto del centro de los datos según la distancia que se está utilizando (a la izquierda la distancia euclídea; a la derecha la distancia de Mahalanobis). Se aprecia que en la distancia euclídea se forman hiperesferas para calcular las distancias, mientras que en la distancia de Mahalanobis se dibujan elipses.

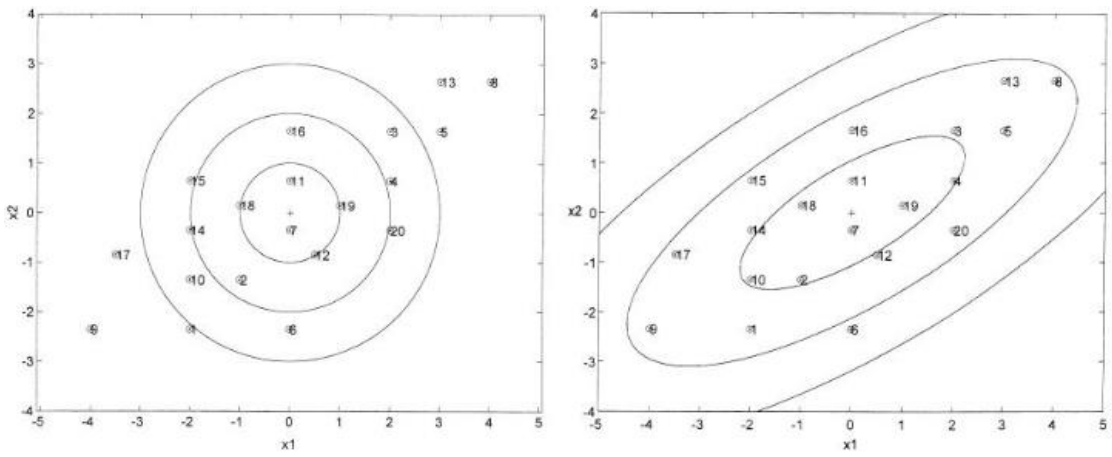

Figura 4: Distancia euclídea y Mahalanobis. (Imagen tomada de [12])

2.3.3 Aplicaciones

La distancia de Mahalanobis es también una distancia muy popular y, en concreto, se ha utilizado en las siguientes aplicaciones:

- **Distancia generalizada de Mahalanobis para una mezcla de datos [13]**: León y Carriere desarrollaron una generalización de la distancia de Mahalanobis para una mezcla de variables nominales, ordinarias y continuas. Ambos autores parten del supuesto de que la distancia estadística entre poblaciones está presente en muchas técnicas de análisis multivariado, con variables continuas. Sin embargo, tratándose de una mezcla de variables discretas y continuas, no existe un modelo de distancia estándar. Esta distancia debe considerar, de acuerdo con los autores, no solo los

diferentes niveles de medición de las variables, sino los diversos tipos de asociaciones entre las variables.

- **Método de monitoreo de la fatiga y ruptura de herramientas de corte [14]**: Wolfstadt JI et al desarrollaron un nuevo método para obtener la distancia de Mahalanobis aplicada a la fatiga y ruptura de herramientas de corte en la industria, con base en imágenes. Los autores establecen que, en comparación con la distancia euclídea, la de Mahalanobis tiene una relación no sólo con la distribución de cada componente del sistema o conjunto, sino con la distribución de los pixeles de la imagen dentro de cada componente de dicho conjunto. De acuerdo con los autores, este método tiene una mayor sensibilidad en la inspección de las condiciones de abrasión en las herramientas, en comparación con los métodos tradicionales.

- **Modelación bioclimática con énfasis en la distancia de Mahalanobis [15]**: Farberr y Kadmon introducen el concepto de la distancia de Mahalanobis a la modelación bioclimática. Ellos afirman que la envolvente climática definida a través de la distancia de Mahalanobis suministra predicciones más precisas de las distribuciones de especies vivientes que las envolventes de los métodos rectilíneos tradicionales.

2.4 k vecinos más próximos (knn)

El algoritmo de los k vecinos más próximos (knn) se trata de un sistema de clasificación supervisado (estimación basada en un conjunto de entrenamientos) que utiliza criterios de vecindad. Knn se basa en la idea de que los nuevos ejemplos serán asignados a la clase a la cual pertenezca la mayor cantidad de ejemplos más cercanos en el conjunto de entrenamiento. Suponiendo K clases que denominamos C_j con i en {1, …, K} en un conjunto de datos X se estima la función de densidad $F(x \mid C_j)$, es decir, es un método de clasificación no paramétrico que estima el valor de la función de densidad de probabilidad o la probabilidad a posteriori de que un elemento x pertenezca a la clase C_j a partir de la información proporcionada por el conjunto de entrenamiento.

Knn es considerado un algoritmo de aprendizaje vago, esto significa que los datos de entrenamiento son almacenados sin ningún procesamiento hasta que llega un nuevo punto en la fase de clasificación. La ventaja principal que supone es debida a la función objetivo, ya que es aproximada localmente con cada nuevo punto, permitiendo que los sistemas de aprendizaje vago puedan simultáneamente resolver múltiples problemas y gestionar exitosamente cambios en el dominio del problema. Esto provoca una serie de desventajas como grandes requerimientos de espacio o computación más intensiva, lo que produce que normalmente sean más lentos en la evaluación, tengan alta susceptibilidad a la maldición de la dimensionalidad, etc.

2.4.1 Algoritmo

El conjunto de entrenamiento está formado por vectores x_i con i ∈ (1 ,…, N) en un espacio multidimensional X. Cada vector está descrito en términos de p atributos, como se muestra en la ecuación 2.3:

$$x_i = (x_{1i}, x_{2i}, \ldots x_{pi}) \in X \quad (2.3)$$

Cada vector x_i tiene asociada una clase C_j de las K clases existentes. La fase de entrenamiento del algoritmo consiste en almacenar los vectores y las clases de los ejemplos de entrenamiento. En la fase de clasificación aparece la constante k definida por el usuario ya qué a cada ejemplo de test (ejemplo sin clase asignada) se le clasifica asignándole la clase que es más predominante en los k ejemplos más cercanos en el conjunto de entrenamiento.

2.4.1.1 Algoritmo de entrenamiento

Al ser un algoritmo de aprendizaje vago, para cada ejemplo en esta fase simplemente se agrega a la estructura de vectores de aprendizaje cada par formado por su ejemplo x_i y su clase c_i.

2.4.1.2 Algoritmo de clasificación

Dado un ejemplo *x* a clasificar y sean $x_1, \ldots x_k$ los k vecinos más cercanos a *x*:

$$\hat{f}(x) \leftarrow argmax_{c \in C} \sum_{i=1}^{k} \delta(c, f(x_i)) \quad (2.4)$$

Siendo δ la distancia, f la función que asigna la clase al ejemplo a clasificar, c la clase a la que esta asignada ese ejemplo y $argmax_{c \in C}$ devuelve el argumento c para el cual la suma es máxima (la clase más frecuente). El valor devuelto por el algoritmo es el valor más común de entre los *k* vecinos más cercanos a x. Para entender más el funcionamiento de la clasificación del algoritmo se muestra el siguiente pseudocódigo:

```
COMIENZO
     Entrada D = {(x1, c1),..., (xn, cn)}
          x = nuevo ejemplo a clasificar
     Para todo objeto ya clasificado (xi, ci)
          Calcular δi = δ(x,xi)
     Ordenar δi(i=1,…,N) en orden ascendente
     Quedarnos con los k casos D_x^K ya clasificados más
     cercanos a x
     Asignar la clase Cj más frecuente en D_x^K
FIN
```

A continuación, en la Figura 1 se observa cómo funciona el algoritmo al clasificar un ejemplo (la estrella roja) con dos clases disponibles y variando el parámetro k (número de vecinos). En ella se aprecia la importancia de la asignación de valores al número de vecinos, ya que asignándole el valor tres, el ejemplo es clasificado como clase “B”, mientras que asignándole el valor 6, el ejemplo es clasificado como clase “A”.

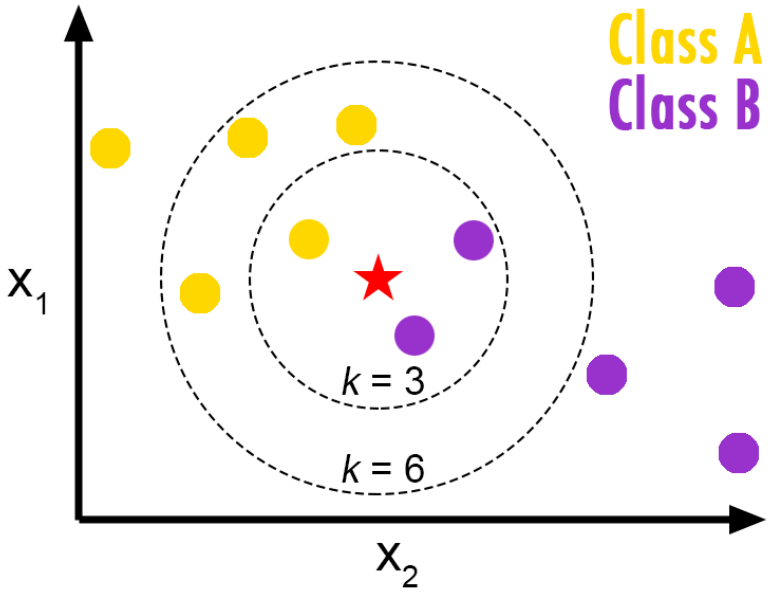

Figura 5. Ejemplo de funcionamiento del algoritmo knn. (Imagen tomada de [16])

2.4.2 Aplicaciones

Knn es un algoritmo que tiene una amplia variedad de aplicaciones en distintas disciplinas[17]:

- **Agricultura:** En general el algoritmo knn se aplica menos que otros modelos de clasificación en campos relacionados con la agricultura. Sin embargo, se ha utilizado para simular las precipitaciones diarias y otras variables climáticas o para la evaluación de inventarios forestales y estimación de variables forestales. En estas aplicaciones se utilizan imágenes por satélite con el objetivo de mapear el uso y cobertura de la tierra. Otras aplicaciones del algoritmo knn en agricultura incluyen la predicción del clima y la estimación de parámetros del agua que se encuentran en el suelo.

- **Finanzas:** La previsión del mercado de valores es una de las tareas financieras más importantes en las que se utiliza el algoritmo knn. La previsión del mercado de valores incluye el descubrimiento de las tendencias del mercado, planificar las estrategias de inversión, identificar el momento para comprar las acciones y qué acciones comprar, predecir el precio de unas acciones sobre la base de las medidas de rendimiento de la empresa y los datos económicos, etc. Otras aplicaciones financieras en las que se utiliza knn son, por ejemplo, comprender y administrar el riesgo financiero, quiebras bancarias, gestión de préstamos, análisis de lavado de dinero, perfiles de clientes bancarios, etc.

- **Medicina:** En esta disciplina nos es muy útil el uso de algoritmos de predicción como knn ya que pueden predecir si, por ejemplo, una persona tiene más riesgo de padecer un infarto basándonos en las mediciones demográficas, dietéticas y clínicas de ese paciente o identificar los factores de riesgo para el cáncer de próstata basándonos en

variables clínicas y demográficas. El algoritmo knn también se ha aplicado para analizar datos de expresión génica de microarreglos donde el algoritmo se ha acoplado con algoritmos genéticos, que se utilizan como herramienta de búsqueda.

3 Diseño

3.1 Introducción

El objetivo principal de este trabajo es el diseño de un nuevo clasificador. En concreto, se pretende realizar un clasificador que determine la clase de un nuevo punto en función de otros puntos similares a él, por esta razón lo denominaremos clasificador por análogos. Este tipo de clasificación es la que realiza knn, por lo que utilizaremos este método como base, pero definiendo una nueva distancia que busque eficientemente los puntos análogos en el conjunto de entrenamiento.

La nueva distancia se basa en la distancia de Mahalanobis, que como se ha descrito anteriormente, utiliza la información de los atributos de los datos mediante la matriz de covarianza, pero vamos a forzarla a actuar en un entorno local del nuevo punto, teniendo por tanto también en cuenta la distancia euclídea al buscar estos ejemplos cercanos. A esta nueva distancia la denominaremos distancia de Mahalanobis local.

A continuación, se detalla el diseño de esta distancia, su implementación y las librerías utilizadas para ello.

3.2 Mahalanobis Local

Recordemos en primer lugar la fórmula de la distancia de Mahalanobis entre dos ejemplos:

$$d_M(x, y) = \sqrt{(x-y)^T \Sigma^{-1} (x-y)} \tag{3.1}$$

Siendo Σ^{-1} la *matriz de covarianzas* del conjunto de datos.

Sobre esta fórmula vamos a realizar una modificación que es la siguiente:

$$d_{ML}(x, y) = \sqrt{(x-y)^T [\Sigma(x)^{-1} + \Sigma(y)^{-1}](x-y)} \tag{3.2}$$

Siendo $\Sigma(x)$ y $\Sigma(y)$ la *matriz de covarianzas* centrada en cada ejemplo. Esta matriz de covarianzas se genera entorno al punto considerado mediante *una nube de puntos* la cual nos proporcionara un entorno local del ejemplo que nos permite buscar similitudes. Para definir estos ejemplos más cercanos pertenecientes a la *nube de puntos* utilizamos la *distancia euclídea.* Con estos ejemplos calculamos la matriz de covarianzas que nos facilitará un entorno en el que buscar similitudes con dicho ejemplo. Cabe destacar que al introducir las covarianzas centradas en los dos puntos involucrados, la nueva distancia sigue siendo simétrica.

Con esta modificación lo que pretendemos al utilizar la matriz de covarianzas de cada ejemplo es la de simular el comportamiento que se produce al utilizar la técnica de *análisis de componentes principales* (PCA) de forma local. PCA se trata de una técnica para describir

un set de datos en términos de nuevas variables (“componentes”) no correladas y así identificar patrones [18]. Esto nos permite reducir o eliminar la redundancia estadística entre los componentes de datos vectoriales de una alta dimensión y representarlos en menos dimensiones sin una perdida significativa de información.

Esta técnica fue inicialmente desarrollada por Pearson a finales del siglo XIX y posteriormente estudiadas por Hotelling en los años 30 del siglo XX. PCA nos permite que las relaciones que se presentan entre variables correladas del conjunto original de variables puedan transformarse en otro conjunto de nuevas variables incorreladas entre sí denominadas componentes principales. Las nuevas variables son combinaciones lineales de las anteriores y se van construyendo según el orden de importancia en cuanto a la variabilidad total que recogen del conjunto de datos. De modo ideal, se busca un menor número de nuevas variables con respecto a las variables originales, que sean combinación lineal de las originales y que estén incorreladas, recogiendo la mayor parte de la información o variabilidad de los datos. Si las variables originales están incorreladas, entonces no tiene sentido realizar PCA en el conjunto de datos.

En la siguiente figura se observa la proyección de las dos componentes principales de un set de datos:

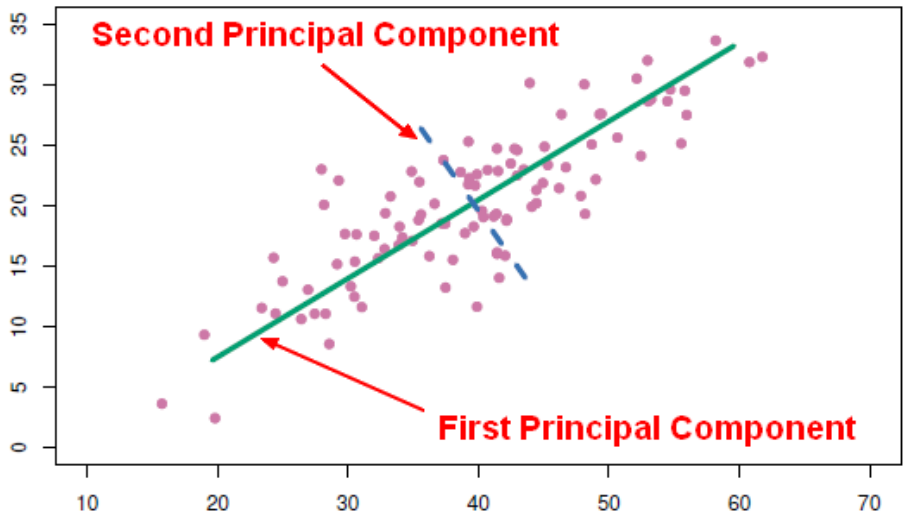

Figura 6: Ejemplo componentes principales tras aplicar PCA a un set de datos (Imagen tomada de [19])

La primera componente principal se trata de la combinación lineal de las variables que tienen la mayor varianza, mientras que la segunda componente principal se trata de la combinación lineal de las variables que tienen la mayor varianza y que no están correladas con la primera componente principal.

Sin embargo, PCA presenta ciertas limitaciones:

- Suposición de linealidad: Se asume que los datos observados son combinación lineal de una cierta base.

- Importancia estadística de la media y la covarianza: PCA utiliza los vectores propios de la matriz de covarianzas y solo encuentra las direcciones de ejes en el espacio de variables considerando que los datos se distribuyen de forma gaussiana.

A la vista de estas limitaciones se ha impulsado el desarrollo de alternativas no lineales a PCA [20]. El término PCA local fue utilizado por primera vez por Braverman (1970) y Fukunaga y Olsen (1971) para denotar el más simple enfoque de clúster PCA que consistía en aplicar el algoritmo *k-medias* u otro tipo de *clustering* a un dataset y calcular los componentes principales de cada clúster por separado. Este efecto es el que pretendemos conseguir con la nueva distancia de Mahalanobis local, donde para cada nuevo punto definimos un entorno local y aplicamos sobre éste la definición clásica de Mahalanobis. Puesto que Mahalanobis es en esencia la distancia euclídea en el espacio de PCA, lo que estamos consiguiendo es una versión de PCA local.

3.3 Implementación

Una vez definido el diseño del nuevo clasificador recurrimos al lenguaje de programación *Python* para realizar la implementación. El lenguaje de programación *Python* se está estableciendo como uno de los lenguajes más populares de computación científica gracias a su naturaleza interactiva de alto nivel y su ecosistema de bibliotecas científicas, convirtiéndolo en una opción muy atractiva para el desarrollo de algoritmos de *machine learning* y el análisis de datos. *Scikit-Learn* se aprovecha de las bondades que nos ofrece *Python* para proporcionarnos implementaciones de calidad de una gran cantidad de conocidos algoritmos de aprendizaje automático, tanto de aprendizaje supervisado como no supervisado, para utilizarlos e integrarlos con el resto del código de una manera asequible[21].

La librería *scikit-learn* nos proporciona la llamada al algoritmo de clasificación knn (*KNeighborsClassifier)* utilizando en él una métrica propia, facilitando en gran medida la implementación. Para la codificación de la nueva distancia local encapsulamos el código en la clase *ClasificadorKnn* en el archivo Clasificador.py disponible en el anexo A.

En esta clase lo primero que hacemos es heredar la clase *BaseEstimator* de la librería *scikit-learn* donde definimos los métodos *fit* y *predict* que son comunes a todos los clasificadores. En el método *fit* se realiza el entrenamiento del modelo, en el caso de knn solo se almacenan los ejemplos de la partición de entrenamiento. Antes de instanciar el clasificador knn de la librería *scikit-learn*, procedemos a calcular las matrices de covarianza de todos los ejemplos del conjunto de entrenamiento utilizando el método *calTrainCovariances*, el cual a su vez llama al método *calCovariencePoint* que calcula la matriz de covarianzas entorno a un ejemplo utilizando para ello un parámetro "n" al que asignamos un valor para fijar la nube de puntos comentada anteriormente. Una vez realizado este cálculo de matrices de covarianza, instanciamos el clasificador knn mediante la clase *KNeighborsClassifier* en el que asignamos a su parámetro "metric" el método *calDistance*. Este método lo utilizará el clasificador cuando calcule la distancia entre dos ejemplos.

El método *predict* se encarga de la clasificación con el clasificador knn de la partición de test del conjunto de datos, siendo en este momento cuando el clasificador utiliza el método *calDistance* que hemos definido como parámetro al instanciarlo. En este método *calDistance* antes de calcular la distancia de Mahalanobis local como indica la fórmula 3.2,

comprobamos si los dos ejemplos poseen su respectiva matriz de covarianza mediante el método *checkCovarienceMatrix*. En el caso que no la posea alguno de los dos ejemplos la calculamos llamando de nuevo al método *calCovariencePoint*.

Puesto que hay que calcular tantas matrices de covarianzas como ejemplos que constituyen el conjunto de datos, el coste computacional de la distancia de Mahalanobis local que hemos definido es bastante elevado, aumentando en gran medida el tiempo de ejecución con respecto a las métricas utilizadas por *scikit-learn*.

3.4 Librerías utilizadas

El lenguaje de alto nivel *Python,* ampliamente utilizado hoy en día, nos es muy apropiado en este particular caso ya que cuenta con diversas librerías de alta utilidad.

3.4.1 NumPy

Es el paquete fundamental de computación científica que nos ha proporcionado numerosas funciones en la implementación realizada, como las operaciones con matrices o el cálculo de las matrices de covarianza [22]. NumPy es la librería encargada de añadir toda la capacidad matemática y vectorial a *Python* haciendo posible operar con cualquier dato numérico o *array* puesto que incorpora desde operaciones elementales como la suma o la multiplicación hasta otras muchas más complejas de álgebra lineal. Además, incorpora herramientas que nos permiten incorporar código fuente de otros lenguajes de programación como *C/C++* o *Fortran* lo que incrementa notablemente su compatibilidad e implementación.

El desarrollo y la principal finalidad del módulo Numpy es la creación y modificación de *arrays* multidimensionales. Un *array* es el término que traslada el concepto matemático de vector o matriz a la programación añadiéndole la noción de almacenamiento en memoria. Estos *arrays* disponen en su interior de una serie de elementos dispuestos en filas y/o columnas dependiendo de la dimensión.

A continuación, se citan las funciones de esta librería utilizadas en este trabajo:

- Numpy.array: Creación de *arrays.*
- Numpy.dot: Realiza la multiplicación entre dos *arrays.*
- Numpy.sqtr: Devuelve la raíz cuadrada de un elemento o *array*.
- Numpy.cov: Calcula la matriz de covarianza a partir de un *array.*
- Numpy.linalg.inv: Devuelve la inversa de una matriz.
- Numpy.random.rand: Crea una matriz a partir de una dimensión dada y la rellena de valores aleatorio entre 0 y 1.
- Numpy.concatenate: Concatena varias *arrays* en una sola.

3.4.2 Scikit-learn

Se trata de una librería que nos ofrece herramientas de *machine learning*, minería de datos y análisis de datos. Nos ha permitido implementar nuestra distancia de Mahalanobis local mediante la llamada a algoritmos de clasificación y además llamar a funciones relacionadas con el aprendizaje automático[23].

A continuación, se citan las funciones de esta librería utilizadas en este trabajo:

- Sklearn.neighbors.KNeighborsClassifier: Implementación del algoritmo de clasificación k vecinos más próximos. Sobre este algoritmo utilizamos la distancia de Mahalanobis local implementada.

- Sklearn.neighbors.KDTree: Implementación de un árbol k-dimensional. Este árbol es utilizado para obtener los ejemplos de la *nube de puntos*.

- Sklearn.model_selection.train_test_split: Divide un conjunto de datos en particiones aleatorias de entrenamiento y test. Utilizado para entrenar los modelos con la partición de entrenamiento y probar su precisión con la partición de test.

- Sklearn.decomposition.PCA: Implementa la técnica de análisis de componentes principales (PCA). Nos es útil para dibujar los gráficos de los conjuntos de datos cuya dimensionalidad es alta.

- Sklearn.model_selection.GridSearchCV: Realiza una búsqueda exhaustiva sobre los valores de los parámetros especificados para un modelo. Lo utilizamos para definir los mejores modelos posibles (con los valores más sensatos para sus parámetros).

3.4.2 Matplotlib

Paquete para generar gráficos en 2 dimensiones. Se ha utilizado para representar los conjuntos de datos [24]. A continuación, se citan las funciones de esta librería utilizadas en este trabajo:

- Matplotlib.pyplot: Proporciona un entorno para la realización de gráficas similar al que nos proporciona el lenguaje de programación *Matlab*.

- Matplotlib.lines.Line2D: Dibuja una línea. Utilizado para la creación de las figuras que indican las clases reales en la leyenda de los gráficos.

- Matplotlib.patches.Patch: Dibuja un rectángulo con un color de fondo y un color de borde. Utilizado para indicar las clases predichas por cada modelo en la leyenda de los gráficos.

4 Pruebas y resultados

4.1 Introducción

En este capítulo describiremos qué experimentos se han utilizado para comparar las distintas métricas que hemos comentado en los anteriores apartados, así como los distintos conjuntos de datos que se han utilizado.

4.2 Descripción y análisis de los conjuntos de datos

Una vez seleccionadas las distancias con las que deseamos experimentar, procedemos a utilizar distintos modelos derivados de aplicar knn con estas distancias. Concretamente utilizaremos tres modelos de knn, uno utilizando la distancia euclídea, otro la distancia Mahalanobis y el último con Mahalanobis local, siendo este último modelo el clasificador por análogos propuesto en el trabajo. Con estos tres modelos definidos, para comprobar su validez y comportamiento se van a utilizar distintos conjuntos de datos.

Para este trabajo hemos utilizado conjuntos de datos extraídos del sitio web *LIBSVM Data* [25]. Se trata de una página en la cual se encuentran distintos conjuntos de datos para problemas de clasificación, regresión, etc. Estos conjuntos de datos se encuentran en un formato *SVM (Support Vector Machine)* que mediante la librería *scikit-learn* se pueden manipular para utilizarlos, siendo parte de esta manipulación la normalización de cada conjunto de datos en el caso de que no lo estuviera ya. Para los experimentos hemos utilizado distintos conjuntos de datos con distintos números de clases, de ejemplos y de atributos.

Estos datos se facilitan de forma matricial donde cada fila representa un ejemplo y cada columna se identifica con un atributo a excepción de la primera columna que representa la clase. El formato SVM nos proporciona los datos en formato *sparse*, es decir, solo los atributos que tienen un valor distinto de 0. Esto se observa en los distintos conjuntos de datos ya que encontramos ejemplos sin ciertos atributos.

Estos datos se encuentran separados en dos particiones, una partición de entrenamiento y otra partición de test, para entrenar el modelo con los datos de entrenamiento y probar su precisión con los de test. Para este proyecto se ha utilizado solo la partición de entrenamiento y se han realizado las particiones comentadas de forma manual utilizando la librería *scikit-learn*.

Para la realización de estos experimentos se han seleccionado 10 conjuntos de datos los cuales son:

- **Heart:** Compuesto por atributos como sexo, edad, cantidad de colesterol en sangre, localización del dolor en el pecho, etc. Es un conjunto de datos binario (dos clases), que indican la presencia o no de una enfermedad cardiaca.

    - Número de ejemplos: 270.
    - Atributos: 13.

- Clases: 2.

- **Glass:** Las clases que lo forman son distintos tipos de cristales y los atributos son elementos químicos que forman parte de la composición de los cristales como sodio, magnesio, aluminio, etc. Esta clasificación de tipos de cristales está motivada por su uso en investigaciones criminales en escenas de un crimen.

  - Número de ejemplos: 214.
  - Atributos: 9.
  - Clases: 6.

- **Liver Disorders:** Conjunto de datos binario que indica si cada ejemplo correspondiente a un único individuo masculino padece un trastorno hepático o no. Los atributos son variables de analíticas de sangre que son sensibles a los trastornos hepáticos que pueden surgir del consumo excesivo de alcohol.

  - Número de ejemplos: 145.
  - Atributos: 5.
  - Clases: 2.

- **Svmguide2:** Conjuntos de datos extraídos de una aplicación bioinformática de la universidad Simon Fraser, Canadá.

  - Número de ejemplos: 391.
  - Atributos: 20.
  - Clases: 3.

- **Svmguide4:** Conjunto de datos extraído de una aplicación sobre señales luminosas de tráfico de la universidad de Ciencias Aplicadas de Berlín.

  - Número de ejemplos: 300.
  - Atributos: 10.
  - Clases: 6.

- **Vowel:** Conjunto de datos basado en el reconocimiento de las onces vocales estables del inglés británico.

  - Número de ejemplos: 528.
  - Atributos: 10.
  - Clases: 11.

- **Vehicle:** Las clases indican si la silueta de un vehículo pertenece a un vehículo de la marca Opel, un vehículo de la marca Sabb, a una furgoneta o un autobús. Los atributos son extraídos de la silueta del vehículo mediante la toma de fotografías.

  - Número de ejemplos: 846.
  - Atributos: 18.
  - Clases: 4.

- **Diabetes:** Conjunto de datos binario que indica si cada ejemplo correspondiente a una mujer de al menos 21 años con herencia india padece diabetes o no. Los atributos son edad, presión arterial, cantidad de glucosa en sangre, etc.

    - Número de ejemplos: 768.
    - Atributos: 8.
    - Clases: 2.

- **Breast-cancer:** Conjunto de datos binarios que indica si cada ejemplo correspondiente a una paciente con un tumor en la mama padece un tumor benigno o maligno. Los atributos son edad, menopausia, localización del tumor en la mama, dimensión del tumor, etc.

    - Número de ejemplos: 683.
    - Atributos: 10.
    - Clases: 2.

- **Sonar:** Conjunto de datos binarios con el objetivo de diferenciar entre señales de sonar rebotadas en un cilindro de metal y aquellas rebotadas en rocas más o menos cilíndricas.

    - Número de ejemplos: 208.
    - Atributos: 60.
    - Clases: 2.

4.3 Experimentos

Nuestros experimentos se centrarán en la comparación de la precisión entre nuestro modelo con nuestra propia métrica implementada y los modelos con las métricas de *scikit-learn*, creando para ello la clase *PruebaDataset* para encapsular todo este proceso de experimentación. Para medir la precisión de estos modelos se evalúa el porcentaje de predicciones correctas (casos en que asignan a un ejemplo la clase real a la que pertenece) sobre el conjunto de test, pero para mostrarlo de una manera más sencilla decidimos mostrar el porcentaje de predicciones erróneas, es decir el error que obtiene el modelo. A continuación, se muestra la fórmula:

$$Error(c, cp) = 1 - \frac{1}{n_{test}} \sum_{i=0}^{n_{test}-1} 1(c_i = cp_i) \qquad (4.1)$$

Siendo cp la clase que se ha predicho, c la clase real a la que cada ejemplo i pertenece y n_{test} el tamaño de la partición de test.

En la clase implementada *PruebaDataset* se crean estos modelos a los cuales se los somete a la técnica de *validación cruzada*, que consiste en realizar varias particiones distintas del conjunto de datos en conjunto de entrenamiento y test, para luego promediar los errores obtenidos por cada partición y elegir el mejor modelo en base a los parámetros. Los parámetros involucrados en estos modelos como es el número de vecinos en los tres modelos

y la *nube de puntos* en nuestra distancia de Mahalanobis local adoptan distintos valores mediante una rejilla de valores que se inicializa antes de todo. Los valores de esta rejilla se fijan de forma básica observando la extensión del conjunto de datos con el que se va a experimentar, y a continuación se realiza una búsqueda de los mejores valores de esta rejilla que deben adoptar los parámetros de los modelos para su uso en la clasificación de la partición de test. Este proceso de búsqueda se realiza gracias a la función *GridSearchCV* que nos proporciona la librería *scikit-learn*. Una vez se encuentran los mejores parámetros, se realiza la clasificación de los ejemplos de la partición de test del conjunto de datos, mostrándonos mediante la función *printData* de nuestra clase el porcentaje de error que ha obtenido cada modelo en la clasificación y los parámetros utilizados para realizar la clasificación. Además, en esta clase se han creado el método *visualizationData* para visualizar cada conjunto de datos que se somete a estos experimentos y el método *visualizationClassification* para observar como el modelo realiza la clasificación de la partición de test del conjunto de datos. En el Anexo B se proporciona el código implementado.

Con este proceso definido utilizamos el entorno de *Jupyter* que nos proporciona sus *Notebooks* [26]. Los *Jupyter Notebook* son documentos altamente útiles para análisis de datos, visualización de datos y *machine-learning*. Contienen tanto código (en nuestro caso código *Python*) como elementos de texto enriquecido, de manera que nos permite introducir análisis y resultados (figuras, tablas) así como fragmentos de código que pueden ser ejecutados en el *Notebook*. Así de esta manera realizamos los experimentos con los distintos conjuntos de datos almacenando todo en un *Jupyter Notebook*.

Una vez completado todo este proceso de experimentación recogimos los resultados obtenidos y realizamos la siguiente tabla donde observamos:

- El conjunto de datos sobre el que se experimentó con los tres modelos mencionados (***Dataset)***.

- El porcentaje de error obtenido en la clasificación de la partición de test con el modelo que utiliza la distancia de Mahalanobis local (***ML (%)***) correspondiente al clasificador por análogos propuesto.

- El porcentaje de error obtenido en la clasificación de la partición de test con el modelo que utiliza la distancia de Mahalanobis global (***MG (%)***).

- El porcentaje de error obtenido en la clasificación de la partición de test con el modelo que utiliza la distancia de euclídea (***E (%)***).

- El número de ejemplos de los que dispone el conjunto de datos (***Ndat***).

- La dimensión o número de atributos que poseen los ejemplos de cada conjunto de datos (***Dim***).

- Número de vecinos utilizados por el modelo con la distancia de Mahalanobis local para realizar la clasificación de la partición de test ($\boldsymbol{K_{ML}}$).

- Número de puntos utilizados en la nube de puntos del modelo con la distancia de Mahalanobis local para calcular la matriz de covarianzas de cada ejemplo ($\boldsymbol{N_{ML}}$).

- Número de vecinos utilizados por el modelo con la distancia de Mahalanobis global para realizar la clasificación de la partición de test (K_{MG}).

- Número de vecinos utilizados por el modelo con la distancia euclídea para realizar la clasificación de la partición de test (K_E).

Dataset	*ML (%)*	*MG (%)*	*E (%)*	*Ndat*	*Dim*	K_{ML}	N_{ML}	K_{MG}	K_E
Heart	**12.3**	17.2	19.7	270	13	9	60	5	5
Glass	**30.7**	38.4	35.3	214	9	1	70	1	1
Liver Disorders	**18.2**	29.5	34	145	5	9	15	11	13
Svmguide2	**22.8**	23.7	26.2	391	20	5	80	3	7
Svmguide4	**35.5**	36.6	46.6	300	10	1	110	1	7
Vowel	**0.6**	1.2	1.2	528	10	1	150	1	1
Vehicle	**21.6**	23.6	29.5	846	18	3	340	5	1
Diabetes	**30.7**	**30.7**	31.1	768	8	15	180	11	9
Breast-Cancer	**1.9**	4.8	2.4	683	10	11	70	5	11
Sonar	26.9	36.6	**12.6**	208	60	1	90	7	1

Tabla 1:Comparativa de error entre modelos.

En las siguientes figuras mostramos el resultado de la clasificación de la partición de test de los tres modelos en el conjunto de datos *Liver Disorders* y en el conjunto de datos *Sonar* de forma gráfica utilizando la función *visualizationClassification*. Para realizar estos gráficos hemos utilizado la técnica de PCA que nos la proporciona la librería *scikit-learn*, para reducir la dimensión de los conjuntos de datos a dos componentes principales y poder así pintarlos en el eje de coordenadas. Para observar de manera más clara la clasificación que realizan los modelos, las clases reales a las que pertenece cada ejemplo son representadas con figuras, mientras que las clases predichas por el modelo son representadas mediante colores, siendo de esta manera más fácil identificar los ejemplos dónde el modelo analizado ha predicho de manera errónea o de manera correcta.

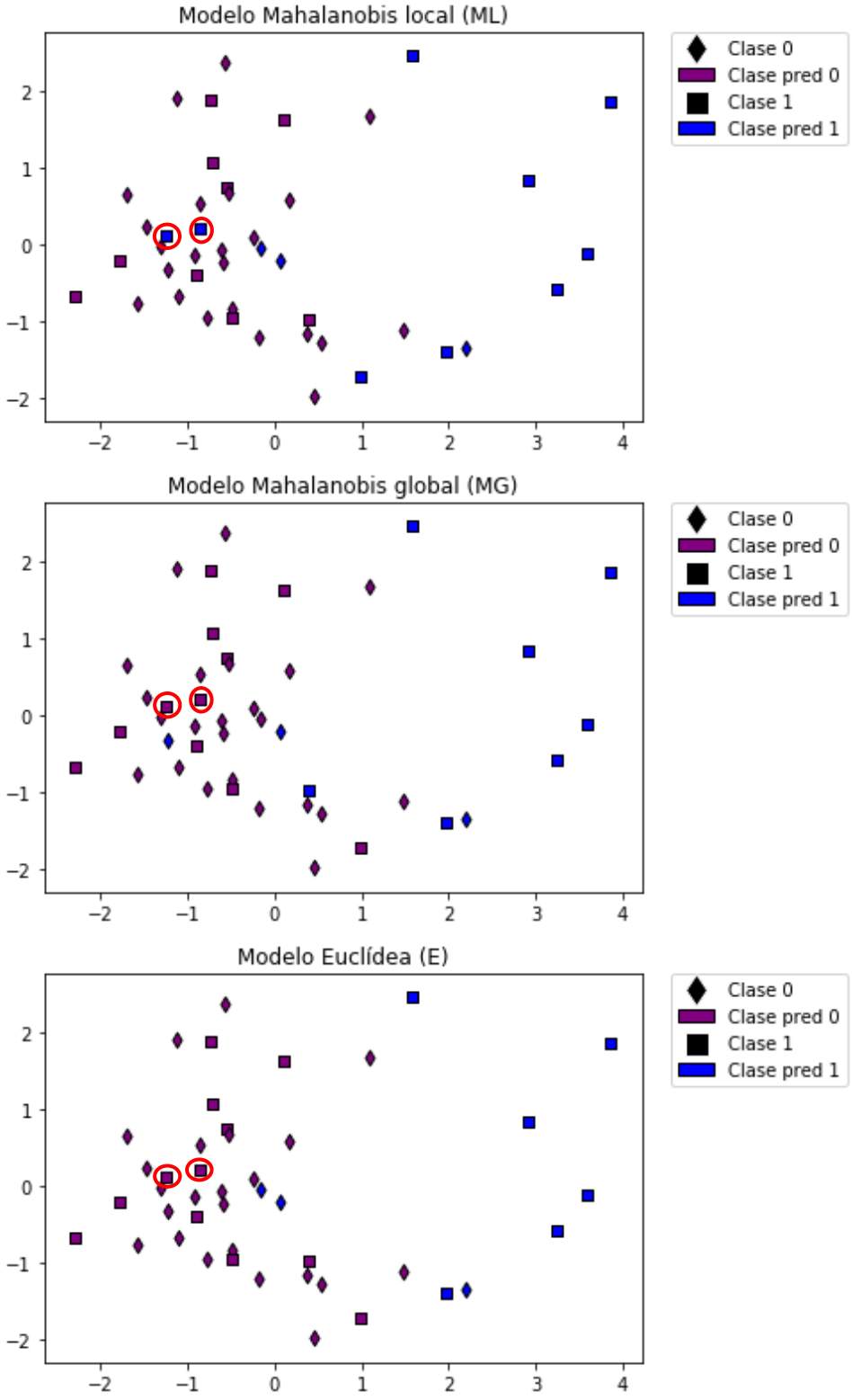

Figura 7: Clasificación de los tres modelos sobre la partición de test del conjunto de datos *Liver Disorders*.

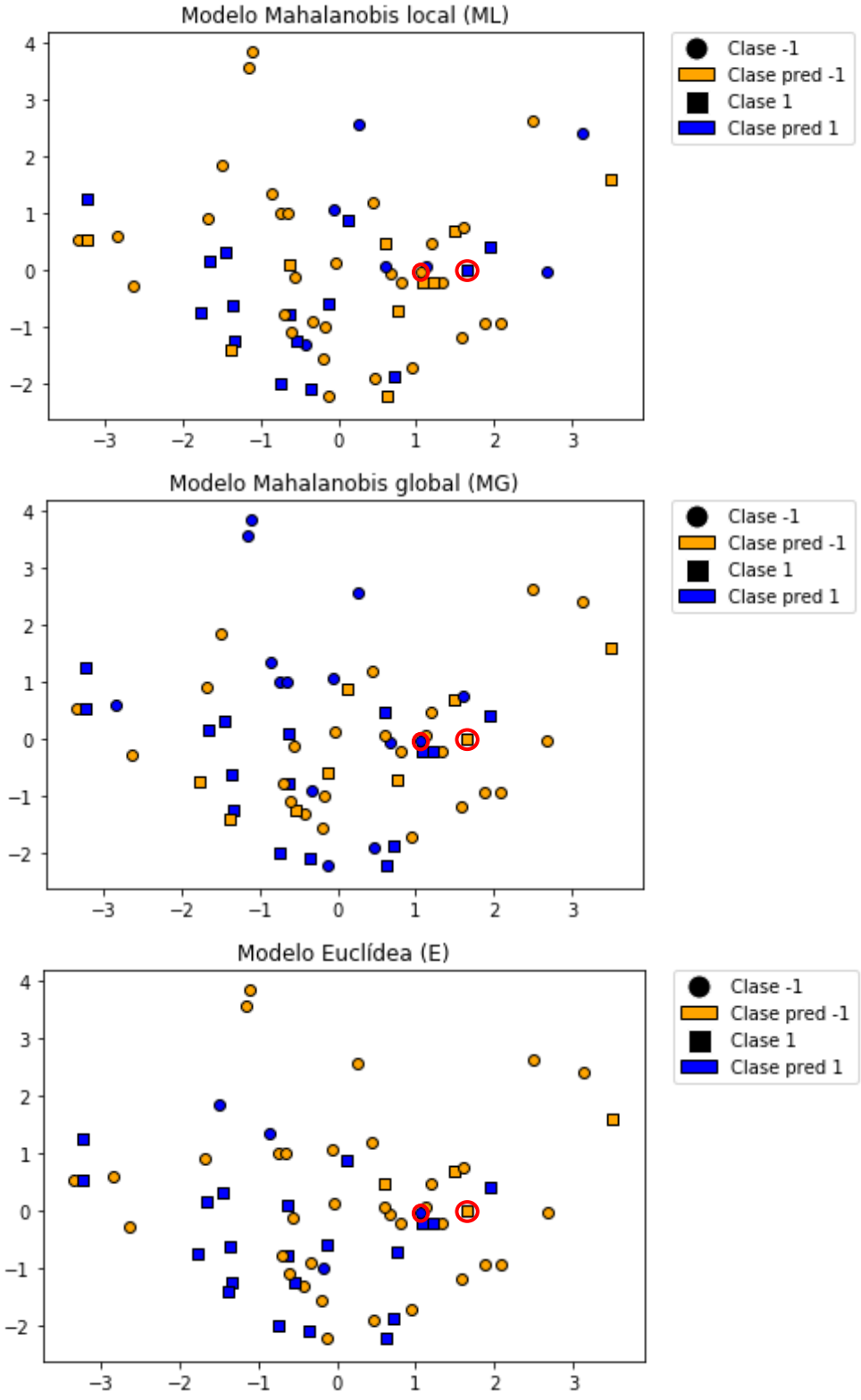

Figura 8: Clasificación de los tres modelos sobre la partición de test del conjunto de datos *Sonar*.

4.4 Conclusiones experimentos

A partir de la tabla de comparativa entre los modelos podemos sacar una serie de conclusiones. Se observa que, en conjuntos de datos como *Heart*, *Liver-Disorders*, *Svmguide2*, *Svmguide4*, *Vehicle* y *Diabetes* el modelo que utiliza Mahalanobis global ya obtiene mejores resultados con respecto al modelo que utiliza la distancia euclídea. Sin embargo, nuestro clasificador por análogos mejora aún más estos resultados como se aprecia en la tabla. En el caso del conjunto de datos *Diabetes* hay que señalar que nuestro modelo obtiene la misma precisión que el modelo con Mahalanobis global, pero en nuestro caso vemos que con un entorno de 180 puntos ya tenemos la información necesaria para igualar al modelo global.

En los conjuntos de datos *Glass*, *Vowel* y *Breast-Cancer* el modelo con Mahalanobis global obtiene peores resultados que el modelo que utiliza la distancia euclídea, por lo que podríamos pensar que el modelo implementado va a obtener también peor precisión. Sin embargo, como se aprecia en la tabla, este modelo de Mahalanobis local mejora además de los resultados del Mahalanobis global, también los resultados del modelo euclídeo.

Se aprecia que en los conjuntos de datos *Glass*, *Vowel* y *Svmguide4* el número de vecinos que utilizan los modelos para realizar la clasificación es uno solamente, es decir solo utilizan el ejemplo más cercano. Este fenómeno se puede deber a que estos conjuntos de datos disponen de varias clases, a diferencia del resto en el que los ejemplos solo pertenecen a dos clases. Al haber más de dos clases, encontrar puntos cercanos de la misma clase puede ser complicado, por lo que el modelo determina que lo más seguro es tomar un único vecino, es decir, asignamos la clase del punto más cercano.

En la Figura 7 se muestran los resultados de clasificación de los distintos modelos probados sobre el conjunto de datos *Liver Disorders*. En este conjunto hemos destacado dos ejemplos de clase 1. Se puede observar que tanto el modelo MG que utiliza la distancia clásica de Mahalanobis como el modelo E, que utiliza la distancia euclídea, clasifican mal este ejemplo. Sin embargo, el modelo ML que utiliza nuestra distancia de Mahalanobis local clasifica este ejemplo adecuadamente aun estando rodeado de ejemplos de la clase 0, por lo que parece que esta distancia no sólo selecciona los puntos más cercanos en el espacio usual (el euclídeo) sino que tiene en cuenta otras propiedades de los ejemplos (como la información de sus atributos) para determinar su semejanza. Hay que señalar que también comete errores en ejemplos que los otros modelos sí clasifican de manera correcta, aunque la precisión de manera global es mayor.

En la Figura 8 se representa la clasificación del único conjunto de datos de los utilizados en los experimentos en el que nuestro modelo implementado obtiene una mejor precisión con respecto al modelo de Mahalanobis global, pero una peor comparándolo con el modelo que utiliza la distancia euclídea. El clasificador por análogos obtiene una peor precisión en este conjunto de datos, aunque se ha resaltado en la figura un par de ejemplos en el que nuestro modelo realiza la clasificación de manera adecuada, mientras que tanto el modelo que utiliza Mahalanobis global como la distancia euclídea lo clasifican erróneamente. Aun así, el modelo de Mahalanobis local no es capaz de mejorar la precisión de manera global, probablemente debido a la estructura particular de este conjunto de datos.

5 Conclusiones y trabajo futuro

5.1 Conclusiones

Este trabajo de fin de grado sirve como conclusión a los estudios realizados en el grado de Ingeniería informática. En dicho trabajo se ha realizado una labor de investigación y se ha utilizado parte del conocimiento adquirido en el grado.

La idea inicial del trabajo era utilizar una medida menos usual como la distancia de Mahalanobis en el algoritmo de clasificación knn, para comparar su eficacia con respecto a otras medidas más tradicionales como la distancia euclídea. Pensamos que sería interesante estudiarlo debido a que la distancia de Mahalanobis utiliza información sobre la correlación entre los ejemplos del conjunto de datos a la hora de calcular la distancia entre dos ejemplos, lo que nos podría ser útil en determinados conjuntos de datos para obtener mayor precisión al realizar las tareas de clasificación. Se han utilizado distintos conjuntos de datos para experimentar con ambos modelos y comparar la precisión que obtienen en la clasificación para representarla en una tabla y extraer conclusiones.

A tal efecto, además de comparar estas distancias, decidimos realizar una implementación de la distancia de Mahalanobis de manera local, definiendo un nuevo modelo de clasificación al que denominamos clasificador por análogos. Para comprobar la eficiencia del modelo propuesto, se comparan los resultados obtenidos frente a los modelos que utilizan la distancia tradicional de Mahalanobis y la distancia euclídea. Todos los experimentos se han realizado en el lenguaje de programación *Phyton*, poco explorado en el grado de Ingeniería Informática, pero particularmente útil en el ámbito del aprendizaje automático gracias a ser un lenguaje interpretado y sobretodo gracias a su librería *scikit-learn*, que facilita la implementación de muchos de los modelos utilizados en esta área.

A la vista de los resultados y del resto del trabajo, las conclusiones más importantes que podemos extraer son las siguientes:

- **Análisis de los resultados obtenidos.** Los resultados obtenidos de los experimentos sobre distintos conjuntos de datos muestran que el modelo que utiliza nuestra distancia de Mahalanobis local obtiene mayor precisión con respecto a los otros modelos en casi todos los conjuntos de datos, con lo cual de esta manera llegamos a la conclusión de que este clasificador que combina tanto las propiedades de la clasificación utilizando la distancia euclídea, como las propiedades de la clasificación utilizando la distancia de Mahalanobis, ofrece una manera alternativa de clasificación con el algoritmo knn, y que cuyo uso en muchos casos es la mejor opción con respecto a la distancia euclídea y la distancia de Mahalanobis.

- **Implementación de la distancia en Python.** Debido a la existencia de distintas librerías de aprendizaje automático en este lenguaje de programación, propiciado por su extendido uso en esta disciplina, decidimos utilizarlo para desarrollar la distancia de Mahalanobis local. Además, su sencillo uso y facilidad para el procesamiento de datos nos incentivó a utilizarlo en este trabajo. *Python* nos proporciona la librería de aprendizaje automático *scikit-learn* que nos ha facilitado en gran medida el trabajo y la librería *NumPy* que también nos lo ha facilitado gracias al procesamiento de los

datos, así como un código que ocupe menos espacio y sea más legible. Por todo esto consideramos la elección de este lenguaje para este desarrollo como un gran acierto.

5.2 Trabajo futuro

Tras la realización de este trabajo quedan una serie de cuestiones que podrían explorarse en trabajos posteriores, las cuales se detallan a continuación:

- **Probar con un mayor número de conjuntos de datos y con conjuntos más grandes.** En este trabajo se han utilizado conjuntos de datos con un tamaño máximo de 850 ejemplos, principalmente debido a que la potencia del equipo en el que se han ejecutado estas pruebas no es la suficiente para realizar pruebas con conjuntos de datos más extensos sin que el tiempo de ejecución sea demasiado elevado. Además, nuestra distancia local es más lenta en ejecución que las distancias ya implementadas. Utilizar más conjuntos también nos permitiría definir en qué tipo de datos esta nueva distancia es más útil.

- **Automatizar el proceso de prueba de parámetros para los modelos.** Para probar la eficacia de los modelos se ha utilizado una rejilla con distintos valores para asignar distintos valores a los parámetros de los modelos (el número de vecinos en todos los modelos y el tamaño del entorno local utilizado en Mahalanobis local), pero esta rejilla se modifica manualmente de forma que si los resultados obtenidos con esos parámetros nos son satisfactorios debemos cambiarlos y volver a ejecutar los modelos. Sería interesante desarrollar una manera de automatizar este proceso.

- **Aplicación de la nueva distancia a otros métodos de clasificación.** En el presente trabajo solo se ha aplicado la nueva distancia implementada en el algoritmo de clasificación de k vecinos más próximos, pero sería interesante aplicar esta nueva distancia que combina la distancia de Mahalanobis y la distancia euclídea con otros algoritmos de clasificación y observar que serie de resultados obtiene.

Referencias

[1] B. Marr, “A Short History of Machine Learning -- Every Manager Should Read”, *Fo rbes*, febrero 2016.

[2] https://trends.google.es/

[3] https://books.google.com/ngrams

[4] C. M. Bishop, “Pattern recognition and Machine Learning”, vol. 1 p. 1-4, 2006.

[5] K. Fuchs, “Machine Learning: Classification Models”, *Medium.com*, marzo 2017.

[6] C. M. Cuadras. “Distancias estadísticas. Estadística Española”, vol. 30 p. 295–378, 1989.

[7] F. M. Hernández Arellano, “El concepto de Distancia y su Aplicación en Estadistica Multivariada”, *Researchgate*, junio 2015.

[8] L. Liberti, C. Lavor, N. Maculan, A. Mucherino, “Euclidean distance geometry and applications” SIAM Review,56 (1):3-69, 2014.

[9] G. M. Crippen and T. F. Havel, “Distance geometry and molecular conformation”, 1988.

[10] E. Abbe, F. Baccelli, A.Sankararaman, “Community Detection on Euclidean Random Graphs”, Junio 2017.

[11] M. Hazewinkel, “Encyclopedia of Mathematics”, *Mahalanobis distance*, enero 2012.

[12] R. De Maesschalck, D. Jouan-Rimbaud, and D.L. Massart. “The mahalanobis distance. Chemometrics and Intelligent Laboratory Systems”, 50(1):1 – 18, 2000.

[13] Leon, A.R. y K.C., Carriere, “A generalized Mahalanobis distance for mixed data”, Journal of Multivariate Analysis, nº92, p. 174-185, enero 2005.

[14] Ji, S.M., et al, “Method of monitoring wearing and breakage states of cutting tools based on Mahalabonis distance features” Journal of Materials ProcessingTechnology, octubre 2002, nº129, p. 114-117.

[15] Farber, O. y R. Kadmon, “Assessment of alternative approaches for bioclimatic modeling with special emphasis on the Mahalanobis distance”, Ecological Modeling, nº160, p. 115-130, febrero 2003.

[16] B.DeWilde, “Classification of Hand-written Digits”, *GitHub.io*, octubre 2012.

[17] S. Bafandeh Imandoust And M. Bolandraftar, “Application of K-Nearest Neighbor (KNN) Approach for Predicting Economic Events: Theoretical Background”, S B Imandoust et al. Int. Journal of Engineering Research and Applications, vol.3 p.605-610, Sep-Oct 2013.

[18] L. Smith, “A tutorial on Principal Component Analysis”, p. 1-27, febrero 2002.

[19] N, gerard, “Data Mining - Principal Component (Analysis|Regression) (PCA)”, *gerardnico.com*, junio 2018.

[20] E. Soria Olivas, J. D. Martín Guerrero, M. Martinez Sober, J. R. Magdalena Benedito, A. J. Serrano López, “HandBook of Research on Machine learning Applications and Trends: Algorithms,Methods, and Techniques”,vol. 1 p. 35, 2009.

[21] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M.Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M.Perrot, É. Duchesnay, “Scikit-learn: Machine Learning in Python”, Journal of machine learning research 12 (Oct), 2825-2830, 2011.

[22] Scypy.org, “NumPy Reference”, 2018, [Online], Available: https://docs.scipy.org/doc/numpy/reference/index.html

[23] scikit-learn.org, “API Reference”,2017, [Online] Available: http://scikit-learn.org/stable/modules/classes.html

[24] J.D. Hunter, “Matplotlib: A 2D Graphics Environment”, *Computing in Science & Engineering*, vol. 3 p. 90-95, 2007.
[25] https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets
[26] https://jupyter-notebook.readthedocs.io/en/stable/

Glosario

knn	K Nearest Neighbors
ML	Mahalanobis local
MG	Mahalanobis global
E	Euclídea
Ndat	Número de datos
Dim	Dimensión de los datos
K_{ML}	Número de vecinos en modelo con Mahalanobis local
N_{ML}	Número de puntos de la nube en modelo con Mahalanobis local
K_{MG}	Número de vecinos en modelo con Mahalanobis global
K_E	Número de vecinos en modelo Euclídeo

Anexos

A Código Clasificador.py

```
"""
Esta clase implementa el modelo de clasificación por análogos,
implementado la distancia de Mahalanobis local.
"""
from  sklearn.base import ClassifierMixin,BaseEstimator
import numpy as np
from sklearn import neighbors as neigh
from sklearn.neighbors import NearestNeighbors,KDTree

class ClasificadorKnn(ClassifierMixin,BaseEstimator):
    """
        Clasificador por análogos.
        Utilizamos las clases ClassifierMixin y BaseEstimator
        para heredar los métodos de fit y predict.

        Atributos
        ---------

        covariancelist: lista con las matrices de covarianza
                        de cada punto.

        datatrain:  datos del conjunto de entrenamiento

        data:   datos donde almacenamos los puntos
                que hemos calculado su matriz.
    """
    covariancelist = []
    datatrain = np.array(())
    data = np.array(())

    def __init__(self, k=None,n=None):
        """
            Método de inicio

            Parámetros
            ----------

            k: número de vecinos.

            n: número de ejemplos para la nube de puntos.
            """
        ClassifierMixin.__init__(self)
        self.k = k
        self.n = n

    def fit(self,Xtrain,Ytrain):
        """
        Método fit del clasificador, en el cual calculamos
        todas las matrices de covarianza de todos los puntos
        de train y además instanciamos el clasificador.

        Parámetros
        ----------
```

```
        Xtrain: ejemplos de la partición de test.

        Ytrain: clases a las que pertenecen los ejemplos de la
        partición de entrenamiento.
        """
        self.calTrainCovariances(Xtrain) #Covarianzas de train
        self.knn = neigh.KNeighborsClassifier(self.k, algorithm='brute',
metric=self.calDistance).fit(Xtrain,Ytrain)

    def predict(self,Xtest):
        """
            Método de prediccion del clasificador, devuelve
            las clases que ha predecido
            según los datos que se les suministra.

            Parámetros
            ----------

            Xtest:  ejemplos de la partición de test.
        """
        return self.knn.predict(Xtest)

    def calTrainCovariances(self,Xtrain):
        """
            Método que crea las matrices de covarianzas de los
            puntos de train.

            Parámetros
            ----------

            Xtrain: ejemplos de la partición de entrenamiento.
        """
        tree = KDTree(Xtrain,metric = 'euclidean')#Creamos arbol
        #con datos train
        self.datatrain = Xtrain
        self.data = Xtrain
        self.covariancelist = []
        for n in range (0,Xtrain.shape[0]):#Calculamos las matrices
        #de covarianza de todos los puntos de train
            self.calCovariencePoint(Xtrain[n],Xtrain,tree)

    def calDistance(self,x1,x2):
        """
            Método que devuelve la distancia de mahalanobis Local
            entre dos puntos.

            Parámetros
            ----------

            x1: punto 1.

            x2: punto 2.
        """
        point = x1 - x2
        self.checkCovarienceMatrix(self.data,x1)#Comprobamos si el
        #punto tiene matriz de covarianza
        self.checkCovarienceMatrix(self.data,x2)
        ind1=self.data.tolist().index(x1.tolist())#Obtenemos
        #índice matriz del punto
        ind2=self.data.tolist().index(x2.tolist())
```

```
        cov = self.covariancelist[ind1] + self.covariancelist[ind2]
        m = np.dot(np.dot(point, cov), point)
        return np.sqrt(m)

    def calCovarienceePoint(self,point,data,tree):
        """
        Método que calcula la matriz de covarianza para un punto y
        la añade a la lista de matrices de covarianza.

            Parámetros
            ----------

            point: punto sobre el cual se quiere calcular su matriz de
            covarianza.

            data: datos para calcular la matriz.

            tree: arbol para encontrar los puntos más cercanos
            al punto que deseamos calcular su matriz.
        """
        dist,indices = tree.query([point], k=self.n+1)
        neigind = indices[0]
        neigborhs = data[neigind]
        cov = np.cov(neigborhs,rowvar=False)
        try:
            self.covariancelist.append(np.linalg.inv(cov))#Añadimos a
            #la lista
        except np.linalg.LinAlgError as err:
            #Si ocurre error de matriz singular introducimos
            #un pequeño error
            long = cov.shape
            cov = cov + 1e-6 * np.random.rand(long[0],long[1])
            self.covariancelist.append(np.linalg.inv(cov))#Añadimos a
            #la lista


    def checkCovarienceMatrix(self,data,point):
        """
            Método que omprueba que el punto tenga su matriz
            de covarianza y si no llama a la función que la crea.

            Parámetros
            ----------

            point: punto sobre el cual se quiere comprobar si tiene
            matriz de covarianza.

            data: datos para comprobar si el punto tiene su matriz, y
            si no para crearla.
        """
        if point.tolist() not in data.tolist():
            #Si el punto no se encuentra añadimos punto a datos de
            #train para el arbol y a los datos globales para luego
            #buscarlo
            newdatatrain =
np.concatenate((self.datatrain,np.array([point])))
            newdata = np.concatenate((data,np.array([point])))
            tree = KDTree(newdatatrain,metric = 'euclidean')
            #Calculamos matriz de covarianza utilizando datos de train
            self.calCovarienceePoint(point,newdatatrain,tree)
            self.data = newdata
```

B Código PruebaDataset.py

```
"""
Clase que implementa la funcionalidad necesaria para probar los
distintos modelos y dibujar los gráficos.
"""
import numpy as np
from Clasificador import ClasificadorKnn
from sklearn.model_selection import GridSearchCV
from sklearn import neighbors as n
from sklearn.model_selection import train_test_split
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
from matplotlib.lines import Line2D
from matplotlib.patches import Patch

class PruebaDataset():
    """
      Prueba conjuntos de datos con tres modelos:

        1. Knn con distancia de Mahalanobis local

        2. Knn con distancia de Mahalanobis global

        3. Knn con distancia euclidea
    """
    def testData(self,X,Y,n_neighbors,cloud_points):
        """
            Método en el cual buscamos los mejores parámetros de
            cada modelo, realizamos una partición del conjunto de datos
            en train y test, realizamos método fit de la partición de
            train y una clasificación de la partición de test y
            finalmente pintamos las gráficas.

            Parámetros
            ----------

            X:  ejemplos del conjunto de datos

            Y:  clases a las que pertenecen los ejemplos del conjunto
                de datos.

            n_neighbors: rejilla con distintos valores para el número
                         de vecinos.

            cloud_points: rejilla con distintos valores para la nube
                          de puntos.
      """
        np.seterr(invalid='ignore')
        #Realizamos partición de test(30%) y train(70%)
        X_train,X_test,Y_train,Y_test =train_test_split(X,Y,
        test_size=0.3)

        clas = ClasificadorKnn()
        knn = n.KNeighborsClassifier(algorithm='brute',
        metric ='mahalanobis')
        knn2 = n.KNeighborsClassifier(algorithm='brute',
        metric ='euclidean')

        parameters = {'k':n_neighbors, 'n':cloud_points}
```

```
    parameters2 = {'n_neighbors':n_neighbors}

    #Buscamos los mejores parámetros y realizamos clasificación
    clf = GridSearchCV(clas, parameters)
    clf.fit(X_train,Y_train)
    self.visualizationClassification(X_test,Y_test,
    clf.predict(X_test),'Modelo Mahalanobis local (ML)')
    self.printData(clf,X_test,Y_test)

    clf = GridSearchCV(knn, parameters2)
    clf.fit(X_train,Y_train)
    self.visualizationClassification(X_test,Y_test,
    clf.predict(X_test),'Modelo Mahalanobis global (MG)')
    self.printData(clf,X_test,Y_test)

    clf = GridSearchCV(knn2, parameters2)
    clf.fit(X_train,Y_train)
    self.visualizationClassification(X_test,Y_test,
    clf.predict(X_test),'Modelo Euclídea (E)')
    self.printData(clf,X_test,Y_test)


def visualizationData(self,X,Y):
    """
        Método en el cual realizamos PCA para mostrar el conjunto
        de datos en dos dimensiones.

        Parámetros
        ----------

        X:  ejemplos del conjunto de datos

        Y:  clases a las que pertenecen los ejemplos del conjunto
            de datos.
     """
    pca = PCA(n_components=2)
    pca.fit(X)
    X = pca.transform(X)
    plt.figure('PCA data')
    plt.title('PCA data')
    plt.scatter(X[:, 0], X[:, 1],c=Y,marker='o',edgecolor = 'K')
    plt.show()


def visualizationClassification(self,X,classes,prediction,title):
    """
        Método en el cual realizamos PCA para mostrar el conjunto
        de test en dos dimensiones.

        Parámetros
        ----------

        X:  ejemplos de la partición de test

        clases: clases reales a las que pertenecen los ejemplos
                de la partición de test.

        prediction: clases predichas por el modelo.

        title:  título de la gráfica.
    """
```

```
        pca = PCA(n_components=2)
        pca.fit(X)
        X = pca.transform(X)
        plt.figure(title)
        plt.title(title)
        mark = ['d','s','p','^','*','x','<','h','>','v','o']
        color = ['purple','blue','yellow','green','red',
        'purple','magenta','grey','gold','salmon','orange']
        legend_elements = []

        for x1,x2,clas,pred in zip(X[:, 0], X[:, 1],
classes.astype(np.int64), prediction.astype(np.int64)):
            plt.scatter(x1,x2, marker=mark[clas],
            color=color[pred],edgecolor = 'K')

        for n1,n2 in
zip(np.unique(classes.astype(np.int64)),np.unique(prediction.astype(np.in
t64))):
            legend_elements.append(Line2D([0], [0],
marker=mark[n1],markersize=13, markerfacecolor='black',color='w',
label='Clase ' + str(n1)))
            legend_elements.append(Patch(facecolor=color[n2],
edgecolor='k',label='Clase pred ' + str(n2)))

        plt.legend(handles=legend_elements,bbox_to_anchor=(1.05, 1),
        loc=2, borderaxespad=0.)
        plt.show()

    def printData(self,clf,X,Y):
        """
            Método en el cual mostramos los mejores parámetros
            seleccionados y el error obtenido en la clasificación
            de la partición de test.

            Parámetros
            ----------

            X:  ejemplos de la partición de test

            Y:  clases a las que pertenecen los ejemplos de la
                partición de test.
            """
        print("Mejores parámetros")
        print(clf.best_params_,"\n")
        print("Error")
        print(1-clf.score(X,Y))
```